Online Caching

Jiazhong Nie
directed by Prof. Manfred Warmuth
SOE UC, Santa Cruz

May 16, 2012
Table of Contents

1 Motivation
2 Caching Problem
3 Master Policies
4 Forced and Optional Policies
5 Deterministic VS Randomized
6 Combining Caches
7 Onlineness Measure
Motivations of this research

- How to measure “onlineness” of sequence? Problem
  - My algorithm works on my data
  - Pass to somebody else and it does not work on their data

**Fix:**
- Design expensive comparator that is computed off-line
- Compare cheap online algorithms against comparator on given sequence
What is a fair comparator?

- Comparator can be based on somewhat “unreasonable” resources
- How to measure generalization performance of online algorithm
  - Batch: split into training and test set
  - Online: how to split?

Two main examples:
- Disk spindown problem
- Caching problem
Table of Contents

1. Motivation
2. Caching Problem
3. Master Policies
4. Forced and Optional Policies
5. Deterministic VS Randomized
6. Combining Caches
7. Onliness Measure
Serve requests using a cache of limited size

Policy
Which cached request to evict
to make room for missed request

12 base policies / experts

- LRU, RAND, FIFO, LIFO, MRU, LFU, MFU, SIZE,
- GDS, GD*, GDSF, LFUDA
Switching between Policies

- State = content of cache
- Switch from cache A to B:
  - charge # of refetched requests needed to switch from A to B
- Partition with switching policies

```
# of misses 14 20 34 = 68
# of refetches 14 20 = 34
```

partition of requests
switching policies
SIZE  GDS  LRU
BestRefetch Curve

\[ \text{BestRefetch}(R) = \text{minimum miss rate achievable by any partition with refetch rate } \leq R \]

miss rate = \# of misses over \# of total requests

refetch rate = \# of refetches over \# of total requests

union of baselines = miss rate of the union of 12 baseline caches
## Table of Contents

1. Motivation
2. Caching Problem
3. **Master Policies**
4. Forced and Optional Policies
5. Deterministic VS Randomized
6. Combining Caches
7. Onlineness Measure
Combining Policies in Expert Setting

- One weight per policy/expert
- Weights updated with loss and share update
- Problem: need to know misses of each expert
- Solution: simulate all policies on virtual cache
  - record ID and size
  - but not the actual data
Two Master Policies

When serving a request

- Hedge+ picks policy *randomly* according to their weights

- Rank-Ideal *deterministically* computes requests’ weighted rank and refetches highest ranked requests
A policy ranks its cached requests according to the order of evicting them (low ranked requests are evicted first).

Rank all of the uncached requests with zero:

<table>
<thead>
<tr>
<th>Requests</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- Evict $r_6$ first
- $r_1$ last
- $r_2$ and $r_4$ not in cache
The Weighted Rank of a Request

The weighted rank of a request \( r \) is the sum of its ranks weighted by different policies:

\[
weighted\_rank(r) = \sum_{i=1}^{N} w_i \ rank_{i,r}
\]

where \( w_i \) is the weight of policy \( i \) and \( \text{rank}_{i,r} \) is the rank of request \( r \) by policy \( i \)
At each trail, RankIdeal

- Refetches as many requests as possible by the descending order of their weighted ranks

- After refetching, the cache is loaded with the Ideal Cache (initial segment of the ordered requests).
RankIdeal is better than Hedge+ which uses the same weight update. Why?
# Table of Contents

1. Motivation
2. Caching Problem
3. Master Policies
4. Forced and Optional Policies
5. Deterministic VS Randomized
6. Combining Caches
7. Onlineness Measure
Forced and Optional Policies

- Two types of caching policies
  - Forced: policy is “forced” to cache missed request
  - Optional: policy “may or may not cache” missed request

- In the original paper, 12 baseline policies are forced
  - Hedge+ follows the setting of baseline policies.

- RankIdeal is optional
  - RankIdeal chooses items according to weighted rank
  - If missed request ranked too low, then not cached

- Is this the reason why RankIdeal outperforms Hedge+?
Examples of Forced and Optional Base Policies

- Optional SIZE: only cache a request if it is smaller than the smallest cached one
- Optional LRU and Forced LRU are the same
  - The missed request is the most recently used one
  - Always cached for both variants
Performance with Forced and Optional Base Policies
Observation

Under optional setting
- Rank-Ideal is still better than Hedge+. Forced/Optional is not the reason of Rank-Ideal’s superiority.
- Curves move down, i.e. lower overall miss rates
- BestRefetch curve gets flatter

New goal
- Use BestRefetch curves to design online algorithms
- Find a set of experts so that curves become flat or increasing
Table of Contents

1 Motivation
2 Caching Problem
3 Master Policies
4 Forced and Optional Policies
5 Deterministic VS Randomized
6 Combining Caches
7 Onlineness Measure
Deterministic vs Randomized Algorithms

- Why RankIdeal is better than Hedge+?
- Rank-Ideal is a deterministic algorithm while Hedge+ is a randomized one.
- (Deterministic version of Hedge+) MaxHedge+:
  - Same update as Hedge+
  - Predicts with the policy of maximum weight
Deterministic VS Randomized

Performance MaxHedge+

- MaxHedge+ and RankIdeal perform similarly and outperform Hedge+

- Deterministic algorithms are better on this data. Randomized algorithm should be better on adversary data.
Does RankIdeal combines the caches well? It is unfair in some sense.

Recall that Rank Ideal combines caches with the rank of requests.

- Cache of 10 requests gives total rank of $\sum_{i=1}^{10} i = 55$
- Cache of 5 requests only gives 15

Denote the score of the $i$-th request in a cache of $n$ requests by $\text{score}(i, n)$. To be fair:

$$\sum_{i=1}^{n} \text{score}(i, n)$$ should be a constant not depending on $n$
Defining \( \text{score}(i, n) \)

- For any function \( f(x) : [0, 1] \rightarrow \mathbb{R} \), the following definition of \( \text{score}(i, n) \) is fair:
  \[
  \text{score}(i, n) = \frac{1}{n} f\left(\frac{i}{n}\right)
  \]

- The fairness can be showed by the fact that for any \( n \):
  \[
  \sum_{i=1}^{n} \text{score}(i, n) \approx \int_{0}^{1} f(x) \, dx
  \]
Choice of $f(x)$

Non-negative monotone increase functions: $f(x) = x^\phi \quad (\phi \geq 0)$
Results

For $\phi \leq 1$, score functions give slightly better results than RankIdeal

What is the best way for combining while ignoring the amount of refetching?
Table of Contents

1. Motivation
2. Caching Problem
3. Master Policies
4. Forced and Optional Policies
5. Deterministic VS Randomized
6. Combining Caches
7. Onlineness Measure
Onlineness Measure

Problem: BestRefetch Curve Overfits

- Random permuted data should not have any onlineness. Curve should not go down.

![Graph showing onlineness measure with refetch rate on the x-axis and miss rate on the y-axis, with two curves representing Week Data Set with Random Permutation and Week Data Set.]
Avoiding overfitting

- Reason: best partition chosen on entire request sequence
  - i.e. training = test set
- Batch learning: split into training and testing set
Online learning: split into even/odd requests

Alternative (not worth it?)
- Per pair, randomly choose one for training/testing
New measure of onlineness

- Run policies on training requests only
- For each $R$, find best partition with refetch rate less than $R$
- Run policies on testing requests only
- Report missrate of the best training partitions on testing requests

Decided by BestRefetch($R$) on training requests

<table>
<thead>
<tr>
<th>SIZE</th>
<th>GDS</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R3</td>
<td>R5</td>
</tr>
<tr>
<td>R7</td>
<td>R9</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R13</td>
<td>R14</td>
</tr>
<tr>
<td>R15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIZE</th>
<th>GDS</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>R4</td>
<td>R6</td>
</tr>
<tr>
<td>R8</td>
<td>R10</td>
<td>R12</td>
</tr>
<tr>
<td>R14</td>
<td>R16</td>
<td>R18</td>
</tr>
<tr>
<td>R20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Measure onlineness by misses on testing requests
BestRefetch with Splitting

How does splitting influence the BestRefetch curve? Miss rate increases since the obligatory miss rate increases.
Miss Rate of Testing Requests

No overfitting to random data: testing miss rate goes up immediately.